Follow the Rules Regarding Concurrency Management

William L. Fithen, Software Engineering Institute [vita[2]]

2005-10-03

Failure to follow proper concurrency management protocols can produce serious vulnerabilities. Concurrent access to shared resources without using appropriate concurrency management mechanisms produces hard-to-find vulnerabilities. Many "functions" that are necessary to use can introduce "time of check/time of use" vulnerabilities.

Description

When multiple threads of control attempt to share the same resource but to not follow the appropriate concurrency protection protocol, then any of the following are possible:

- Deadlock: one or more thread may become permanently blocked [Johansson 05[3]].
- Loss of information: saved information is overwritten by another thread [Gong 03[4], Pugh 99[5], Manson 01[6], Manson 05[7]].
- Loss of integrity of information: information written by multiple threads may be arbitrarily interlaced [Gong 03[8], Pugh 99[9], Manson 01[10], Manson 05[11]].
- Loss of liveness: imbalance in access to shared resources by competing threads can cause performance problems [Gong 03[12], Pugh 99[13], Manson 01[14], Manson 05[15]].

Any of these can have security implications, sometimes manifest in apparent logic errors (decisions made based on corrupt data).

Competing "Systems" (Time of Check/Time of Use)

This is the most frequently encountered subclass of concurrency-related vulnerabilities. Many of the defects that produce these vulnerabilities are unavoidable due to limitations of the execution environment (i.e., the absence of proper concurrency control mechanisms). A common mitigation tactic is to minimize the time interval between check and use, but a more effective tactic is use a "check, use, check" pattern that can often detect concurrency violations, though not prevent them.

Applicable Context

All of the following must be true:

- Multiple "systems" must be operating concurrently.
- At least two of those systems must use a shared resource (e.g., file, device, database table row).
- At least one of those systems must use the shared resource in any of the following ways:
  - Without using any concurrency control mechanism. This includes the situation where no such mechanism exists, such a conventional UNIX filesystems, causing corruption or confusion.

2. daisy:320 (Fithen, William L.)
3. #refs
4. #refs
5. #refs
6. #refs
7. #refs
8. #refs
9. #refs
10. #refs
11. #refs
12. #refs
13. #refs
14. #refs
15. #refs

- Using the right concurrency control mechanism incorrectly. This includes situations like not using a consistent resource locking order across all systems (e.g., in databases), causing deadlocks.
- Using the wrong concurrency control mechanism (even if it used correctly). This includes situations where a give resource may support multiple concurrency control mechanisms that are independent of one another (e.g., UNIX lockf() and flock()), causing corruption or confusion.

These defects are frequently referred to as time of check/time of use defects because APIs providing access to the resource neither provide any concurrency control operations nor perform any implicit concurrency control. In this case, a particular condition (e.g., availability of resource, resource attributes) is checked at one point in time and later program actions are based on the result of that check, but the condition could change at any time since no concurrency control mechanism guarantees the condition did not change.

Competing Threads within a "System" (Races)

The second largest class of concurrency-related vulnerabilities is generated by defects in the sharing of resources such as memory, devices, or files. The defect may be a design error associated with the concurrency control mechanisms or with an implementation error such as not correctly using those mechanisms. Caching errors can be considered a member of this class.

Strictly speaking, signal handling defects are not concurrency defects. Signal handlers are invoked preemptively in the main thread of the process. Therefore, signal handlers are not really concurrently executed. However, from the programmer's viewpoint, they mostly feel like concurrent execution, so we classify them here, at least for now.

Applicable Context

All of the following must be true:

- A "system" must have multiple concurrently operating threads of control.
- Two or more of those threads must use a shared data object, device, or other resource.
- At least one thread must use the shared resource without using the appropriate concurrency control mechanism correctly (or at all).

Impacts Being Mitigated

- Impact #1:
  - **Minimally:** None.
  - **Maximally:** Deadlock: one or more threads may become permanently blocked.
- Impact #2:
  - **Minimally:** None.
  - **Maximally:** Loss of information: saved information is overwritten by another thread.
- Impact #3:
  - **Minimally:** None.
  - **Maximally:** Loss of integrity of information: information written by multiple threads may be

arbitrarily interlaced.

- Impact #4:
  - **Minimally:** None.
  - **Maximally:** Loss of liveness: imbalance in access to shared resources by competing threads can cause performance problems.

Security Policies to be Preserved

- Policy #1
  - Threads must not deadlock.
- Policy #2
  - Information must not be lost.
- Policy #3
  - Information must not be corrupted.
- Policy #4
  - Acceptable performance must be maintained.

How to Recognize this Defect

- Concurrency defects are extremely difficult to recognize. There is no general purpose approach to finding them.

Mitigation Advice

To Engineers:

- **Efficacy:** INFINITE
- The appropriate concurrency control mechanism must be used in the *conventional* way (assuming there is one).

To Engineers:

- **Efficacy:** LOW
- Where no concurrency control mechanism is available, seek to minimize the interval between the time of check and the time of use. Technically this does not correct the problem, but it can make the error much more difficult to exploit.

References

[Bishop 96]	Bishop, Matt & Dilger, Mike. "Checking for Race Conditions in File Accesses." *Computing Systems 9*, 2 (1996): 131-152.

[Gong 03]	Gong, Li; Ellison, Gary; & Dageforde, Mary. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation* (2nd Edition). Boston, MA: Addison-Wesley, 2003.
[Johansson 05]	Johansson, Olof& Torvalds, Linus. *Fix possible futex mmap_sem deadlock*. http://linux.bkbits.net:8080/linux-2.6/cset@421cfc11zFsK9gxvSJ2t__FCmuUd3Q (2005). What is a futex[17] anyway?
[Manson 01]	Manson, J. & Pugh, W. "Core semantics of multithreaded Java," 29-38. *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*. Palo Alto, California, USA, 2001. New York, NY: ACM Press, 2001. DOI= http://doi.acm.org/10.1145/376656.376806[18].
[Manson 05]	Manson, J.; Pugh, W.; & Adve, S. V. "The Java memory model," 378-391. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Long Beach, California, USA, January 12-14, 2005. New York, NY: ACM Press, 2005. DOI= http://doi.acm.org/10.1145/1040305.1040336.
[Pugh 99]	Pugh, W. "Fixing the Java memory model," 89-98. *Proceedings of the ACM 1999 Conference on Java Grande*. San Francisco, California, USA, June 12-14, 1999. New York, NY: ACM Press, 1999. DOI= http://doi.acm.org/10.1145/304065.304106[20].
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.
[VU#132110]	Rafail, Jason. *Vulnerability Note VU#132110: Apache HTTP Server vulnerable to DoS race condition in the handling of short-lived connections*. http://www.kb.cert.org/vuls/id/132110 (2004).

SEI Copyright

17. http://ds9a.nl/futex-manpages/futex4.html

18. http://doi.acm.org/10.1145/376656.376806

20. http://doi.acm.org/10.1145/304065.304106

1. http://www.sei.cmu.edu/about/legal-permissions.html

Fields

Name	Value
Copyright Holder	SEI

Fields

Name	Value
is-content-area-overview	false
Content Areas	Knowledge/Guidelines
SDLC Relevance	Implementation
Workflow State	Publishable